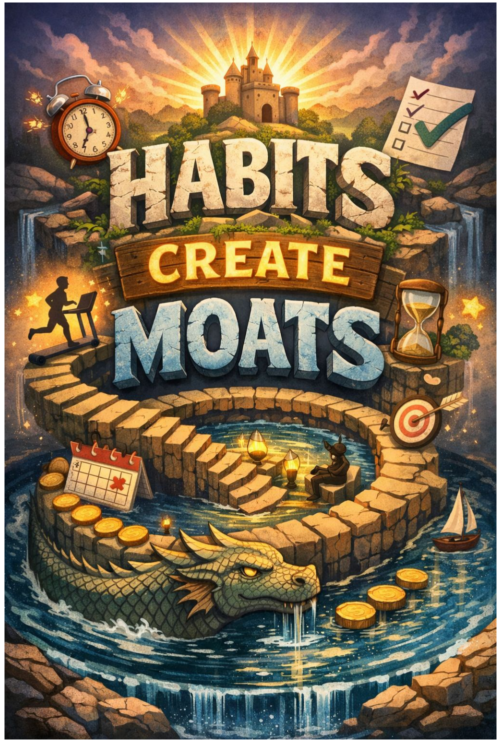
More Context Engineering

Beyond Prompt Engineering: Architecting the Full Context Stack
for Reliable, Production-Grade AI Systems

John Whaley, Jan Jannink CS 224G - 1/22/2026

Administrative Details

- Attendance secret code! Habits Create Moats
  - https://canvas.stanford.edu/courses/221239/quizzes/184860
- Due on Friday
  - Individual check-in form
  - Sprint 1 Planning form
- Mentor matching has started
- Today:
  - More on Context Engineering
  - Data Strategy



Learning Objectives

By the end of this lecture, you should be able to:

Distinguish Context vs. Prompt Engineering

Define context engineering as the broader architecture of information supply, distinct from query phrasing.

Enumerate LLM Call Components

Identify the full stack beyond the user message: system prompts, conversation history, tool definitions, and parameters.

Identify Application-Layer Levers

Determine which context elements (memory, RAG, uploads, tools) are controllable by the application developer.

Design Practical Context Plans

Construct a context strategy that balances information richness with token budget constraints.

Motivation: Prompting Is Not Enough

The quality of LLM output depends heavily on provided context. Consider this progression:

✗ **LEVEL 1: BAD PROMPT, BAD CONTEXT**

"Is giving blood good for you?"

Generic query, zero personalization. Result: Generic WebMD-style advice.

? **LEVEL 2: BETTER PROMPT, BAD CONTEXT**

"Does giving blood help with elevated SHBG levels?"

Specific scientific query, but stateless. Result: Theoretical answer, no specific advice.

✓ **LEVEL 3: BETTER PROMPT, BETTER CONTEXT** **TARGET STATE**

"Does giving blood help with elevated SHBG levels?
[Context Attached: Historical blood panels (2023-2025), donation dates, relevant gene testing results]"

Result: Highly personalized analysis comparing specific lab values against donation timing.

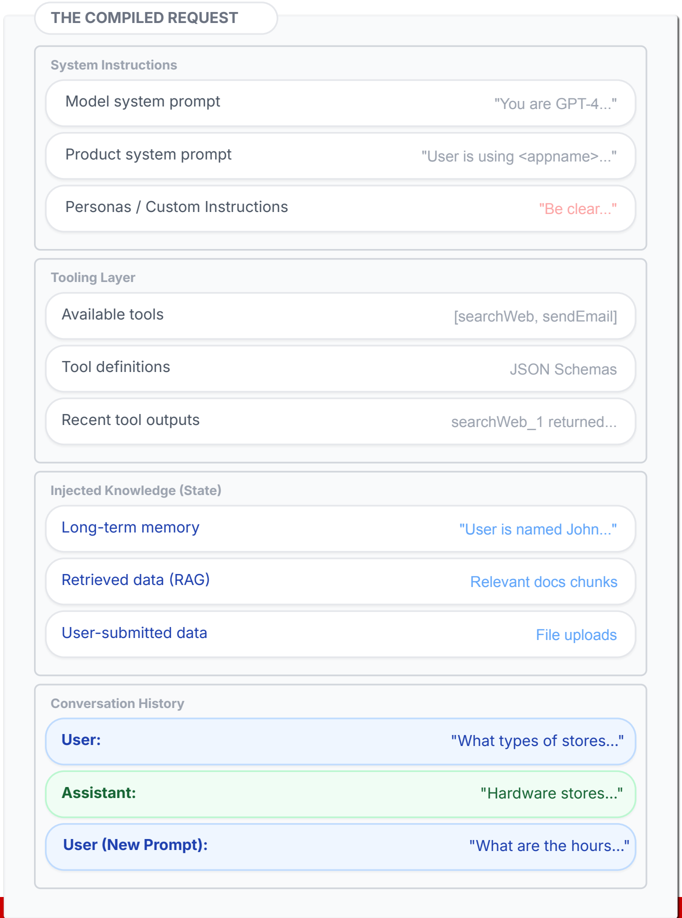
Key Takeaway: Better Prompt + Better Context = **Dramatically Better Results**

SECTION 02

What Goes Into an *LLM Call?*

From a simple chat box to a compiled request.
We'll walk through the layers of the context stack next.

The Context Engineering Model



THE COMPILED REQUEST

System Instructions

Model system prompt	"You are GPT-4..."
Product system prompt	"User is using <appname>..."
Personas / Custom Instructions	"Be clear..."

Tooling Layer

Available tools	[searchWeb, sendEmail]
Tool definitions	JSON Schemas
Recent tool outputs	searchWeb_1 returned...

Injected Knowledge (State)

Long-term memory	"User is named John..."
Retrieved data (RAG)	Relevant docs chunks
User-submitted data	File uploads

Conversation History

User:	"What types of stores..."
Assistant:	"Hardware stores..."
User (New Prompt):	"What are the hours..."

The "Simple Chat" Illusion

What users see as a simple text box is actually the tip of an information iceberg. The application **compiles** this stack for every single turn.

1 **System & Persona Layer**
Defines identity and behavior boundaries. Often composed of model defaults + app logic + user preferences.

2 **Injected State (The "Meat")**
The stateless model knows nothing about you. Memories, RAG docs, and files must be injected afresh every time.

3 **Parameters (Invisible)**
Temperature, Max Tokens, Logit Bias—these govern *how* the stack is processed.

CS224G Insight: Your application logic determines 90% of what the model "knows" before it generates a single token.

Call Parameters & The Context Window

Hidden Control Levers

Temperature 0.0 - 2.0

Controls randomness/creativity. Low temp (0.0) = deterministic; High temp (1.0+) = more creative but less stable.

Max Tokens Integer

Hard limit on generation length. Does *not* limit input size, only output. Crucial for cost control and latency.

Logit Bias Map<Token, Float>

Surgical precision tool. Force the model to avoid or favor specific tokens (words). Often used for classification consistency.

The Token Budget Economy

The "Rule of Thumb"

1 Token ≈ 0.75 Words

(1,000 tokens ≈ 750 words)

Models see tokens, not words. Context limits are strict hard stops.

When the Context Window Fills Up:

The Compression Trade-off

Frontends must summarize history or truncate older messages. You gain space but lose **fidelity**.

The "Lost in the Middle" Phenomenon

Performance degrades as context fills up. Key instructions buried in the middle of a 100k+ token window are often ignored.

CS224G: Lecture 6

System Prompts & Instruction Layers

14:00–18:00

The Three Strata of Instructions

1. Model System Prompt

Immutable

The baseline identity baked in by the provider (OpenAI, Anthropic).

```
"You are GPT-4, a helpful assistant developed by OpenAI..."
```

2. Product System Prompt

Dev Control

Your application's specific framing. Defines the tool's role in your product.

```
"The user is using {AppName} to analyze financial data..."
```

3. Personas / Custom Instructions

User Control

User-defined preferences layered on top.

```
"Be concise. Answer in JSON. Prefer python for math."
```

Engineering Implications

The Underrated Lever: Personas

Most frontends bury this setting, but for power users and complex apps, allowing users to define **global invariants** is crucial.

Instead of repeating "format as markdown" in every prompt, bake it into the persona layer once. This reduces prompt noise and improves compliance.

Best Practices for System Prompts:

- ✓ **Sandwiching**
  Place critical instructions at both the start (System) and end (User) of the context window to combat "lost in the middle."
- ✓ **Separation of Concerns**
  Keep security guardrails in the Product Prompt (Layer 2) so users cannot easily override them in Layer 3.

SECTION 03

Injected *Knowledge*

Where most people think "context" lives:
Memories, Retrieval, and Uploads.

Injected Knowledge

> ***Crucial Mental Model:*** *The LLM is a stateless "brain-in-a-jar." It does not remember anything. Memory is a function of the application layer, not the model layer.*

1. Preserved Memories

Long-term state preserved about the user across sessions.
e.g., "User is named John, lives in CA, teaches at Stanford, prefers concise code."

2. Retrieved Data (RAG)

Data fetched from connected datastores before the call. The model doesn't "search"; the infrastructure searches and injects.

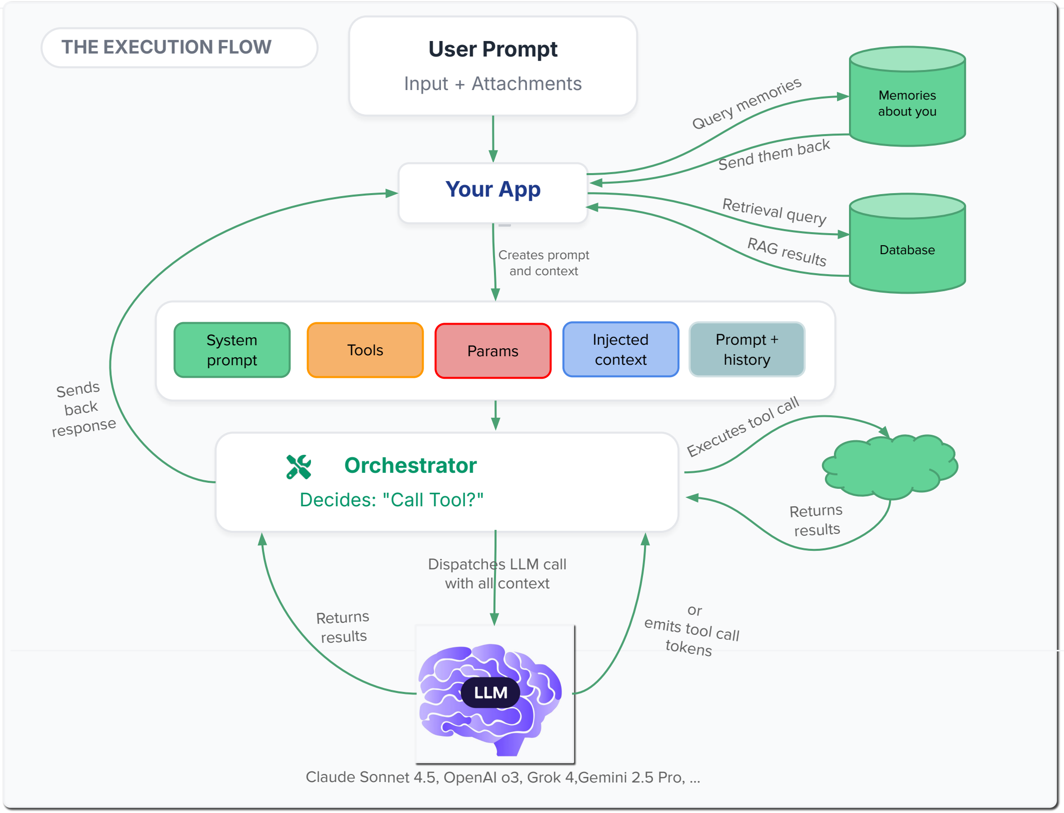
3. User-Submitted Data

Ad-hoc files or "project" attachments.e.g., "Analyze this PDF."

→ **Architecture Example**
Customer Support Agent: App searches internal KB for query topic → Retrieves top 3 articles → Injects text into context → Model generates response.

Tools, Orchestrators & Agentic Loops

22:00–28:00



Tools are Programmable Context

Tools aren't magic. They are standard functions (API calls, scripts) wrapped in a schema the LLM can "read."

The "Brain-in-a-Jar" Reality

The LLM does not execute code or search the web. It emits **tokens** indicating intent. The **orchestrator** (your app) does the actual work.

AGENTIC WORKFLOW

The Loop

Orchestrator runs tool → Output becomes new context → Feed back to LLM → Repeat until done.

What You Can Control

The *Context Engineering* Checklist

While much of the LLM stack feels opaque (model weights, training data), these **five specific levers** are directly in your control as an application builder.

Builder's Tip

"Don't just tweak the prompt. Engineer the inputs that surround it."

- ○ **1. Personas / Custom Instructions**
  Define global invariants here (e.g., "Always return JSON"). Don't waste tokens repeating this in every prompt.
- ○ **2. Available Tools**
  Curate your toolkit. Giving a model 50 tools confuses it. Give it the 5 it actually needs for the task.
- ○ **3. RAG / Datastore Connections**
  You control the search quality. Which databases are connected? How are chunks retrieved? This is a huge performance lever.
- ○ **4. User-Submitted Data**
  Encourage file uploads. Explicit context ("Read this PDF") beats implicit retrieval every time.
- ○ **5. Your Prompt**
  Still matters! But treat it as the final instruction, not the entire context payload.

In-Class Discussion

EXERCISE

The Token Budget Crisis

Your long-running agent chat is hitting the **128k context limit**. Performance is degrading.
What is your eviction policy?

1 **Compression vs. Eviction**

Do you summarize history (lossy compression) or drop oldest turns (eviction)? Why?

2 **The "Important" Bits**

What specific pieces of context must be **pinned** and never dropped? (e.g. system instructions, tool definitions)

3 **Re-retrieval Strategy**

If you drop a file from context, how does the model get it back if needed later?

FORMAT

- Discuss
- 4 Minutes
- Share with class

Context Engineering Recap

1 **Context Engineering > Prompt Engineering**

The prompt is just the tip of the spear. The surrounding context (injected knowledge, history, system instructions) often determines performance more than the phrasing of the question.

2 **Context is a Compiled Artifact**

Don't think of it as "chatting." Think of it as your application **compiling** a massive text file (history + RAG + tools + system prompts) on every single turn.

3 **State Lives in the App, Not the Model**

The LLM is a stateless brain-in-a-jar. All "memory" (user preferences, project files, past turns) is retrieved and injected by your infrastructure.

4 **Master the Five Levers**

You have control over: `Personas`, `Tools`, `RAG`, `Uploads`, and the `Prompt`. Use all five, not just the last one.

> *"Think of the LLMs as a new, genius Ph.D.-level team member... but one for whom it is **always** their first day at work."*
>
> — **Andy Bromberg**